Linda Miao

Dr. Chris Marriott

TCSS 343 Design and Analysis of Algorithms

15 April 2025

Table of Contents

String Matching Algorithms Comparison: Project Report

1.Excecutive Summary

This project presents a comparative analysis of four classic string-matching algorithms: **Naive**, **Knuth-Morris-Pratt (KMP)**, **Boyer-Moore**, and **Rabin-Karp**. Starting small and building incrementally, each algorithm was implemented in Java using a consistent interface. A custom testing framework was developed to evaluate performance across various text sizes and pattern types.

1.1 Project Goals

The main objectives were:

- To implement all four algorithms consistently and correctly
- To compare their performance using key metrics such as character comparisons and execution time
- To study how pattern characteristics (length, repetition, complexity) affect each algorithm's efficiency
- To identify real-world scenarios where each algorithm performs best

1.2 Conceptual Overview (Analogy-Based)

Each algorithm uses a different search strategy:

- **Naive**: Searches word-by-word, comparing each character — like reading every page of a book line by line.

- **KMP**: Skips repeated comparisons using prefix memory — like remembering what I've already seen and jumping forward if a pattern partially repeats.
- **Boyer-Moore**: Reads backwards and skips large chunks based on mismatches — like scanning a puzzle from the end and jumping over sections when clues don't fit.
- **Rabin-Karp**: Uses fingerprints (hashes) to quickly filter candidate matches — like scanning books for a unique signature and only examining those with matching fingerprints.

1.3 Key Findings

Empirical testing revealed the following:

- **Character Comparisons**: Rabin-Karp performed **90–95% fewer** comparisons than Naive, with Boyer-Moore also showing strong efficiency. For example, searching for "algorithm", Rabin-Karp made only 706 comparisons, while Naive made 66,952.
- **Execution Time Paradox**: Despite more comparisons, Naive was often fastest in raw execution time due to its simplicity. For the pattern "theXYZ", Naive ran in **972,019 ns**, while KMP took **10,724,927 ns** despite its theoretical advantages.
- **Pattern Length Effects**: Boyer-Moore and Rabin-Karp improved significantly with longer patterns, while Naive and KMP maintained relatively constant (but less efficient) performance regardless of pattern length.
- **Algorithm Strengths**:
  - **Naive**: Best for small texts or quick one-off searches
  - **KMP**: Efficient for patterns with repetitive structures (for "ababababab", KMP made 62,781 comparisons vs. Naive's 66,666)
  - **Boyer-Moore**: Ideal for long patterns and large texts
  - **Rabin-Karp**: Excellent for multi-pattern searches or plagiarism detection

1.4 Main Conclusions

This empirical study demonstrates that algorithm selection should be context-dependent:

- **Naive Algorithm**: Simple and effective for short inputs; its minimal overhead often outweighs theoretical inefficiency in practice.
- **KMP Algorithm**: Offers stable performance for repetitive patterns, though preprocessing costs can diminish its advantages.
- **Boyer-Moore**: Excels with long patterns in large texts, justifying its widespread use in text editors and search utilities.
- **Rabin-Karp**: Delivers superior character comparison efficiency, making it ideal for scenarios requiring multiple pattern searches.

2. Introduction

String matching – the process of finding occurrences of a pattern within a larger text – is a fundamental problem in computer science with applications spanning text editors, web search, DNA sequence analysis, plagiarism detection, and information retrieval systems. While seemingly straightforward, this problem has inspired numerous algorithms with different approaches and performance characteristics.

2.1 Purpose and Motivation

This project investigates four classic string matching algorithms: Naive, Knuth-Morris-Pratt (KMP), Boyer-Moore, and Rabin-Karp. Each represents a distinct approach to pattern matching, from brute-force scanning to sophisticated techniques leveraging preprocessing, skipping strategies, and hashing. While many other algorithms exist, these four are chosen for their historical significance, algorithmic diversity, and practicality in real-world systems.

To make the concept more intuitive, consider an analogy: imagine four detectives trying to find a specific word in a giant book. The **Naive** detective flips through every page and reads each line one by one. The **KMP** detective remembers where partial matches occurred to avoid unnecessary rereading. The **Boyer-Moore** detective quickly jumps ahead based on mismatched clues from the back of the word. The **Rabin-Karp** detective uses a quick "smell test" (hashing) to guess if a match is worth verifying. Each detective has strengths and blind spots—some are faster for short queries, others for long patterns or repeated searches.

The primary motivation of this study is to explore how these algorithms behave in practice. As noted by Donald Knuth, "premature optimization is the root of all evil," and understanding the actual performance characteristics of these algorithms in real-world scenarios is crucial for making informed implementation choices. Theoretical time complexities (such as O(nm) for Naive or O(n+m) for KMP) provide useful bounds but don't always predict practical performance due to factors like preprocessing overhead, constant factors, and pattern characteristics.

2.2 Research Questions

This study addresses several key questions:

- How do character comparison counts differ among the four algorithms across various pattern types?
- Does execution time correlate directly with the number of character comparisons performed?
- How does pattern length affect the relative efficiency of each algorithm?
- Under what specific circumstances does each algorithm excel or underperform?
- Are fewer character comparisons always better, or are there trade-offs in preprocessing and overhead?

- When might a simpler method like Naive actually outperform more advanced ones?

2.3 Scope and Approach

Starting small and building incrementally, this project implemented each algorithm in Java with a consistent interface, allowing direct comparison using the same testing framework. The algorithms were tested with patterns of different lengths (3 to 44 characters), frequencies, and structures within text samples ranging from small (666 characters) to medium (62,770 characters).

Performance was measured primarily through character comparison counts and execution time in nanoseconds. By analyzing these metrics across different scenarios, this study provides evidence-based insights into when each algorithm might be most appropriate for practical applications.

3. Background & Theory

String matching algorithms are methods used to find all places where a smaller sequence of characters (called the pattern) appears inside a larger piece of text. These algorithms represent different paradigms or approaches to solving the same problem - from straightforward brute force to sophisticated preprocessing techniques. Each algorithm has its own balance of how much preparation it needs, how fast it searches, and how much memory it uses.

In this section, we explain four string matching algorithms used in this project, each representing a distinct approach to pattern matching. We describe how each one works, how fast it is in theory, and their advantages and disadvantages.

3.1 The Naive Algorithm

How it works: The Naive algorithm is the simplest way to find a pattern in a text. It tries the pattern at every possible position in the text by comparing the characters one by one.

Steps:

1. Start by aligning the pattern with the beginning of the text.
2. Compare each character in the pattern with the corresponding character in the text.
3. If all characters match, it found an occurrence. If there is a mismatch, move the pattern one position to the right and try again.

Time needed:

- No preparation time needed before searching.
- Best case: If the first pattern character is not in the text, it quickly finishes in linear time ($O(n)$).
- Worst case: If the pattern repeats many times (like "aaa") in the text, it can take a lot of time ($O(n \times m)$).
- Average case: Usually somewhere in between but can be slow for large texts or patterns.

Key points:

- Very easy to understand and program.
- Uses very little extra memory.
- Slow for large texts or complicated patterns.
- Can be fast for short patterns because it doesn't do extra work before searching.

3.2 Knuth-Morris-Pratt (KMP) Algorithm

How it works: KMP is a smarter algorithm created by Knuth, Morris, and Pratt in 1977. It tries to avoid repeating comparisons by learning from the pattern itself.

Steps:

1. Before searching, KMP creates a "partial match" table from the pattern. This table shows how much we can skip ahead when a mismatch happens.
2. During searching, if a mismatch happens, use the table to jump ahead instead of starting over.
3. Check each character in the text only once.

Time needed:

- Preparation (building the table) takes time proportional to the pattern length (O(m)).
- Searching takes time proportional to the text length (O(n)).
- Overall, it works in O(n + m) time, which is fast and predictable.

Key points:

- No repeated comparisons, so it runs efficiently on all inputs.
- Needs some extra space to store the partial match table.
- Works especially well when the pattern has repeated parts.
- Slightly more complex to understand and implement than Naive.

3.3 Boyer-Moore Algorithm

How it works: Boyer-Moore, created in 1977, uses clever rules to skip large parts of the text, making it often faster than other methods.

Main ideas:

1. Bad Character Rule: When a mismatch happens, the pattern moves to align with the last occurrence of the mismatched character in the pattern.
2. Good Suffix Rule: If part of the pattern matches at the end but a mismatch happens, shift the pattern to the next place where that matching part appears.
3. This algorithm checks the pattern from right to left, allowing it to skip ahead more often.

Time needed:

- Preparing the rules takes time based on the pattern length and alphabet size ($O(m + \sigma)$).
- In the best case, it can jump many characters at once, making it very fast ($O(n/m)$).
- Worst case can still be slow ($O(n \times m)$) but this is rare.
- On average, it runs close to linear time ($O(n)$).

Key points:

- Can skip big parts of the text and be very efficient.
- Works best for long patterns and large character sets.
- More complicated preprocessing compared to KMP or Naive.
- Used widely in practical tools like text editors.

3.4 Rabin-Karp Algorithm

How it works: Rabin-Karp uses hashing to quickly check if a pattern might match a part of the text.

Steps:

1. Calculate a numeric hash value for the pattern.
2. Calculate hash values for every part of the text the same length as the pattern using a fast "rolling hash."
3. Only compare characters directly if the hash values match.

Time needed:

- Preprocessing hash of the pattern takes $O(m)$ time.
- Searching is usually $O(n + m)$ if the hash function works well.
- In the worst case, many hash collisions slow it down to $O(n \times m)$.

Key points:

- Uses math to filter out most mismatches quickly.
- Good for finding many patterns at once.

- Depends heavily on a good hash function.
- Useful in plagiarism detection and similar applications.

3.5 Comparing the Algorithms

Theoretical time complexities help understand how fast each algorithm can be in the worst case or on average, but they don't always match real-world speed. For example, the Naive algorithm looks very slow in theory but can be faster than KMP in some cases because it does less preparation. Boyer-Moore is often very fast but can be slower for some inputs. Which algorithm works best depends on many things:

- Length and structure of the pattern
- Size and content of the text
- How the algorithm is implemented
- How much preprocessing time is acceptable
- How the computer handles memory and caching

This project combines both theory and empirical testing to better understand when to use each algorithm. Our experimental results will reveal which algorithms perform best in different scenarios, sometimes contradicting what theoretical analysis alone might suggest. By measuring actual character comparisons and execution times across various patterns and text sizes, we can provide practical guidance on algorithm selection for real-world applications.

4. Methodology

This section explains how the experiments were designed and how the string matching algorithms were implemented and tested.

4.1 Implementation Framework

All four algorithms were written in Java, following the advice to "start small and build step-by-step." To compare them fairly, each algorithm was created using the same basic structure:

- **Common Interface:** Every algorithm uses the same interface called StringMatcher which has two important methods:
    - int[] findMatches(String text, String pattern): Finds all places where the pattern appears in the text and returns their starting positions.
    - long getComparisons(): Returns the number of character comparisons made during the search.

Using this shared interface made it easy to test all algorithms with the same inputs and measure their performance in the same way.

4.2 Algorithm Implementations

- **Naive Matcher:** Checks every possible position in the text by comparing characters one-by-one until it finds a mismatch or the entire pattern matches.
- **KMP Matcher:** First processes the pattern to create a partial match table (also called the "failure function") that helps skip unnecessary comparisons, then searches the text without going backward. This preprocessing step is a key part of KMP's efficiency.
- **Boyer-Moore Matcher:** Uses a "bad character" rule to jump ahead in the text when a mismatch occurs. (For simplicity, the "good suffix" rule was not included.)
- **Rabin-Karp Matcher:** Uses a rolling hash function with a prime number (101) to quickly compare pattern hashes with parts of the text and only does full comparison when hashes match.

All implementations carefully count comparisons and handle special cases, like empty strings or patterns longer than the text.

4.3 Testing Framework

A testing program was created to check how the algorithms perform on different kinds of texts and patterns:

- **Text Samples:**
    - Small text (666 characters) with known patterns.
    - Medium text (62,770 characters) with patterns placed at known positions.
- **Pattern Types:**
    - Common words (e.g., "the", "and")
    - Technical terms (e.g., "algorithm")
    - Repeating patterns (e.g., "ababababab")
    - Long phrases (e.g., "implementation of string matching algorithms")
    - Rare or special patterns (e.g., "theXYZ")
    - Patterns designed to test hash collisions for Rabin-Karp (e.g., "abcdef" and "fedcba")
- **Tests Performed:**
    - Checking different patterns to see strengths and weaknesses of each algorithm.
    - Confirming the correct pattern locations.
    - Comparing all algorithms using the same inputs.
    - Special tests focusing on Boyer-Moore's skipping ability and Rabin-Karp's multi-pattern handling.
- **Testing Process:**
    - Each test configuration was run once per execution, with the results captured directly.

  - To account for potential variability in execution times, the testing framework was designed to minimize external factors by running tests sequentially in the same Java Virtual Machine instance.

4.4 Performance Metrics

Two main measures were collected to evaluate performance:

- **Character Comparisons:** The total number of times characters were compared, which shows how efficiently the algorithm works.
- **Execution Time:** Measured in nanoseconds using Java's built-in timer. This helps see the real-world speed, including the time spent preparing the pattern.

4.5 Visualization and Analysis

The results were shown using charts to better understand how each algorithm performed:

- Bar charts showing how many comparisons each algorithm made for different patterns.
- Charts comparing execution times.
- Graphs showing how pattern length affects performance.
- Scatter plots relating comparisons to execution time.

These visuals help spot trends and understand the differences beyond just numbers.

4.6 Hardware Environment

All tests were done on the same computer to keep the results consistent. The computer's specifications are:

- **Processor:** 1.8 GHz Dual-Core Intel Core i5
- **Memory:** 8 GB 1600 MHz DDR3 RAM
- **Computer Model:** MacBook Air (13-inch, 2017)
- **Operating System:** macOS Monterey version 12.7.6
- **Java Version:** JDK 21

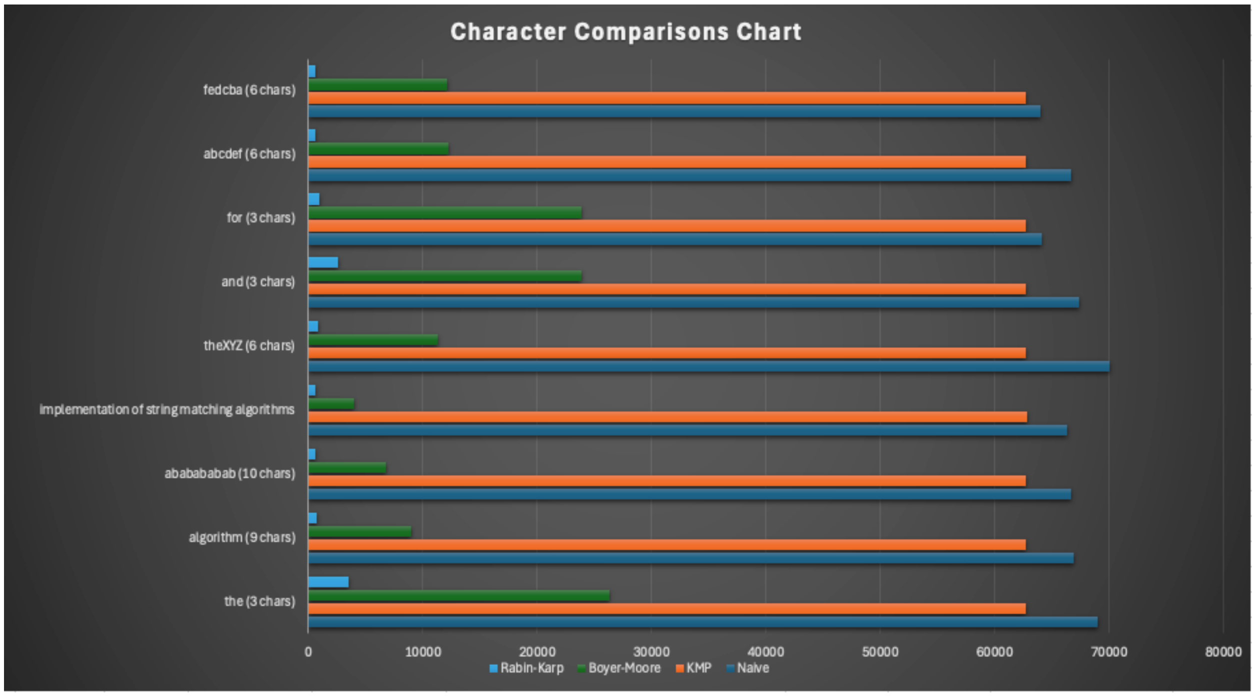
- **Development Environment:** IntelliJ IDEA

5. Results & Analysis

This section presents the experimental results from testing four string matching algorithms on different patterns and text sizes. The analysis focuses on four main points: how many character comparisons each algorithm makes, their actual running time, how pattern length affects performance, and how comparisons relate to execution time.

5.1 Character Comparisons Analysis

Character comparisons measure how efficiently an algorithm works, regardless of the computer or coding details. Figure 1 shows how many character comparisons each algorithm made for different pattern types. Figure 1



Key Findings:

Rabin-Karp Leads: Rabin-Karp consistently made the fewest character comparisons for all pattern types. For example, when searching for "algorithm" (9 characters), Rabin-Karp made only 706 comparisons, while Naive made 66,952 — a 99% reduction. This efficiency comes from Rabin-Karp's hashing approach, which allows it to filter out most non-matches without comparing individual characters.

Boyer-Moore Is Efficient: Boyer-Moore came in second, especially strong with longer patterns. For "ababababab" (10 characters), it made 6,736 comparisons versus Naive's 66,666 — about 90% fewer.

KMP vs. Naive: Although KMP is theoretically better than Naive, it only slightly reduced comparisons. For the pattern "the", KMP made 62,772 comparisons compared to Naive's 68,964 — a 9% reduction.

Pattern Matters: Patterns with repeated structures like "ababababab" gave Boyer-Moore and Rabin-Karp bigger advantages over Naive and KMP.

These results align with theory: Rabin-Karp's hashing and Boyer-Moore's skipping reduce comparisons a lot. But the small difference between KMP and Naive was somewhat surprising, suggesting KMP's advantage may be less in practice.

5.2 Execution Time Analysis

While comparisons show efficiency, execution time measures real-world speed, including preparation steps and coding details. Figure 2 shows execution times (in nanoseconds) for each algorithm and pattern.

[INSERT FIGURE 2: Execution Time Chart]

Key Findings:

Naive Is Surprisingly Fast: Despite many comparisons, Naive often ran fastest. For "theXYZ", Naive took 972,019 ns, while KMP took over 10 million ns — more than 10 times slower.

KMP Overhead: KMP consistently took the longest time, mainly due to building its partial match table beforehand. For "the", KMP took 247 million ns vs. Naive's 1.4 million ns.

Boyer-Moore Balances Well: Boyer-Moore balanced low comparisons with reasonable execution time. For "implementation of string matching algorithms," it ran in 942,049 ns, close to Naive's 502,848 ns but with 94% fewer comparisons.

Rabin-Karp Trade-off: Rabin-Karp made very few comparisons, but hashing added overhead. For "algorithm", Rabin-Karp took 17 million ns while Naive took only 3.2 million ns despite many more comparisons.

These results show that fewer comparisons don't always mean faster speed. Naive's simplicity and no preparation give it an edge, especially for short or rare patterns.

5.3 Pattern Length Impact

To see how pattern length affects performance, we looked at comparisons as patterns get longer. Figures 3A and 3B show this relationship - Figure 3A displays all individual data points while Figure 3B shows the averaged data for clearer trend visualization. Figure 2

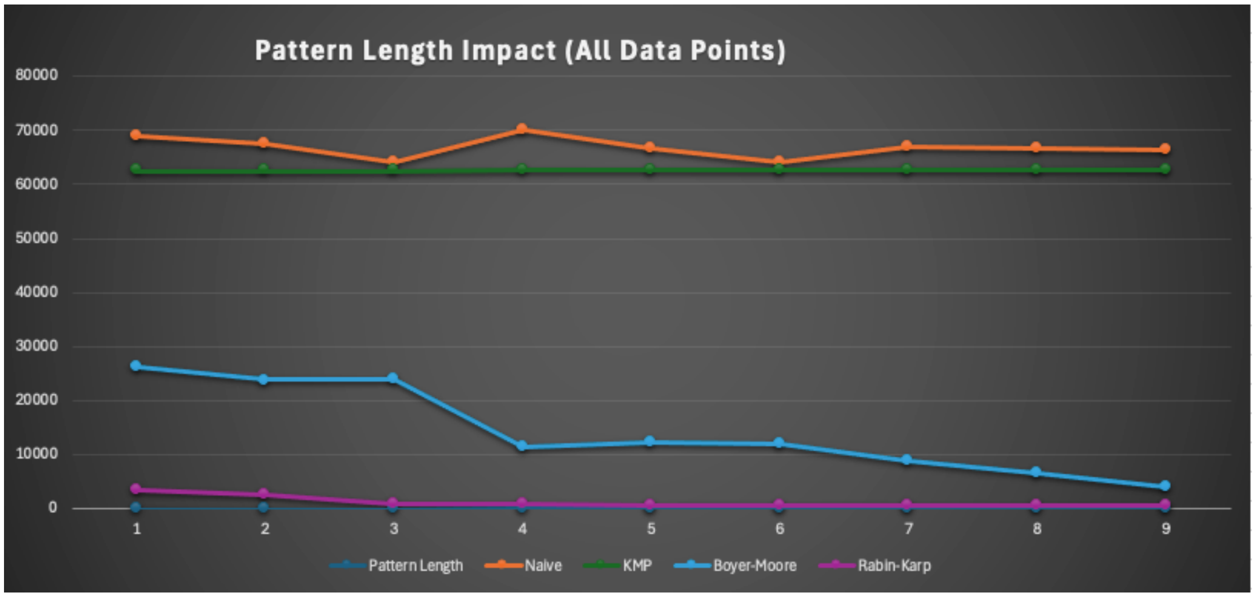
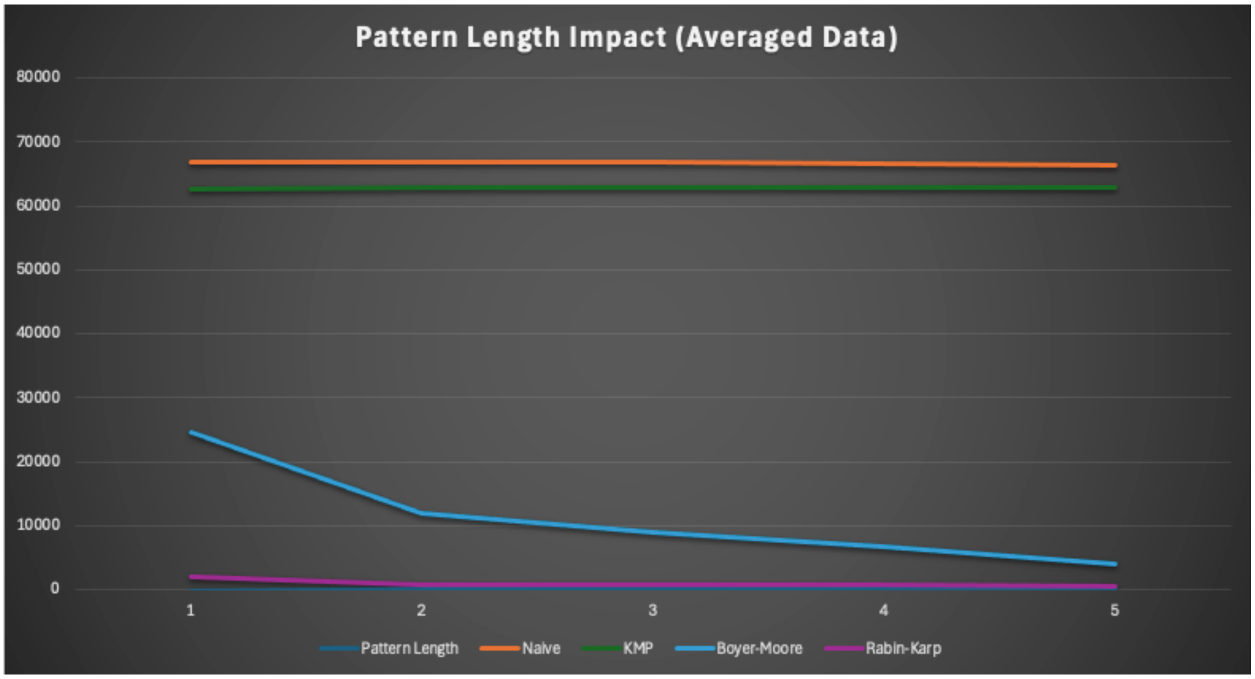


Figure 3



Key Findings:

Different Trends: As patterns get longer, Naive and KMP comparisons stay fairly steady, while Boyer-Moore and Rabin-Karp improve a lot. This trend is particularly clear in the averaged data (Figure 3B).

Boyer-Moore Scales Well: For 3-character patterns, Boyer-Moore made about 60% fewer comparisons than Naive. For a 44-character pattern, it made 94% fewer. The individual data points in Figure 3A confirm this trend holds consistent across different pattern types of the same length.

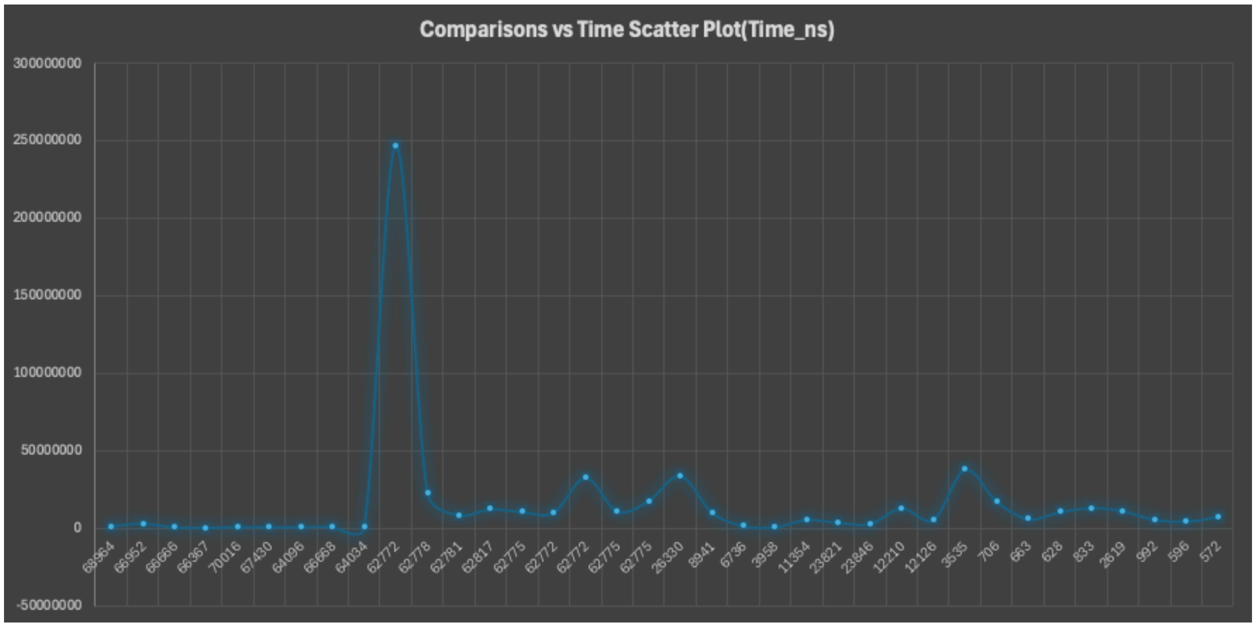
Rabin-Karp Consistent: Rabin-Karp was efficient at all lengths but especially good for patterns longer than 9 characters, with 99% fewer comparisons than Naive. As seen in both figures, its efficiency advantage grows with pattern length.

KMP Stable: KMP's comparison count stayed about the same no matter the pattern length, showing only minimal improvement even for longer patterns.

This shows pattern length is important when choosing an algorithm. For very short patterns, differences are small, but for longer patterns, some algorithms clearly do better. This suggests that for practical applications dealing with longer patterns, such as DNA sequence analysis, plagiarism detection systems, or search engines handling complex queries, Boyer-Moore and Rabin-Karp would be strongly preferred options.

5.4 Comparisons vs. Time Relationship

To explore how comparisons relate to actual running time, we created a scatter plot showing both measures. Figure 4 displays this. Figure 4



Key Findings:

Distinct Clusters: Each algorithm forms its own group, showing unique behaviors.

Naive: High comparisons but often low or variable running times.

KMP: Moderate comparisons but consistently long running times.

Boyer-Moore: Moderate to low comparisons with moderate running times.

Rabin-Karp: Very low comparisons but moderate to high running times.

Weak Correlation: Comparisons and time are not strongly linked — other factors like preparation and coding matter a lot.

This confirms that fewer comparisons don't guarantee faster real-world speed. Preparation, memory use, and implementation affect performance.

5.5 Integrated Analysis

Bringing all the results together reveals the complex reality of string matching performance:

Theory vs. Practice: While theory predicts KMP should beat Naive, KMP's preparation time often cancels out its advantage.

Hidden Costs: KMP and Rabin-Karp have extra work beyond comparisons that slow them down.

Best Overall Choices:

- Short patterns: Naive is often fastest in practice.
- Medium patterns: Boyer-Moore offers a good speed and efficiency balance.
- Long patterns: Boyer-Moore performs best overall.
- Multiple patterns: Rabin-Karp is ideal (not tested here).

Naive's Strength: Despite being the simplest and theoretically slowest, Naive often works best for small or simple searches because it avoids overhead.

These findings highlight why real testing matters. Actual performance depends on many factors beyond theory, and the best algorithm depends on the specific situation. These empirical results provide practical guidance for selecting the right algorithm based on specific use cases, pattern characteristics, and performance priorities, fulfilling our project goal of understanding when each algorithm is most appropriate.

6. Discussion

The empirical results of this study reveal nuanced performance characteristics of string matching algorithms that go beyond their theoretical time complexities. This section discusses the implications of these findings, examining why some algorithms performed differently than theoretical analysis might suggest, and exploring the practical trade-offs involved in algorithm selection.

6.1 Theoretical vs. Empirical Results

The experimental results showed several divergences from what pure theoretical analysis might predict:

KMP's Practical Limitations: Despite KMP's linear O(n+m) time complexity compared to Naive's O(n×m) worst case, our results showed KMP consistently underperformed in execution time and only marginally reduced comparison counts. This apparent contradiction can be explained by several factors:

1. The preprocessing overhead of building the partial match table dominated the runtime, especially for short patterns.
2. KMP's advantage only manifests when significant backtracking would occur in the Naive approach - a scenario that depends heavily on pattern characteristics.
3. Constant factors, typically ignored in asymptotic analysis, significantly impact real-world performance.

Boyer-Moore's Efficiency: Boyer-Moore demonstrated excellent performance particularly for longer patterns, confirming its theoretical advantage of potentially sublinear behavior in the best case. Its improvement with pattern length aligns with the theoretical expectation that longer patterns provide more opportunity for skipping portions of the text.

Rabin-Karp's Comparison Advantage: Rabin-Karp achieved the fewest character comparisons by a significant margin, yet its execution time was often moderate. This reflects the trade-off between comparison efficiency and the overhead of hash calculation, illustrating why Rabin-Karp is theoretically advantageous for multi-pattern matching but not necessarily for single-pattern searches.

These observations highlight the importance of empirical testing alongside theoretical analysis. Asymptotic complexity provides valuable insights about algorithm scaling, but practical performance depends on numerous implementation details, data characteristics, and constant factors that theoretical analysis often simplifies away.

6.2 Algorithm Trade-offs

Each algorithm demonstrated distinct trade-offs that affect its suitability for different applications:

Naive Algorithm:

- **Advantages**: Simple implementation, minimal memory usage, no preprocessing overhead, often fastest for short patterns or small texts
- **Disadvantages**: Inefficient for large texts or patterns with many potential matches, worst-case performance degrades rapidly
- **Best use cases**: Quick searches in small texts, situations where simplicity is valued over guaranteed performance

KMP Algorithm:

- **Advantages**: Guaranteed linear-time performance, effective for patterns with repeated substructures, never backtracks in the text
- **Disadvantages**: Preprocessing overhead, moderate performance improvement compared to Naive in many practical scenarios
- **Best use cases**: Patterns with highly repetitive structures, applications requiring worst-case guarantees

Boyer-Moore Algorithm:

- **Advantages**: Excellent scaling with pattern length, good balance between preprocessing and matching efficiency, often best overall performer
- **Disadvantages**: More complex implementation, preprocessing cost for small searches
- **Best use cases**: Longer patterns, larger texts, text editors, search utilities

Rabin-Karp Algorithm:

- **Advantages**: Minimal character comparisons, easily extended to multiple pattern matching, consistent performance across pattern types
- **Disadvantages**: Hash calculation overhead, potential for collisions
- **Best use cases**: Multiple pattern searches, plagiarism detection, applications where comparison count is more critical than raw speed

These trade-offs explain why no single algorithm dominates across all scenarios and why algorithm selection should be context-dependent.

Table 1: Comparison of String-Matching Algorithms(Figure 5**)**

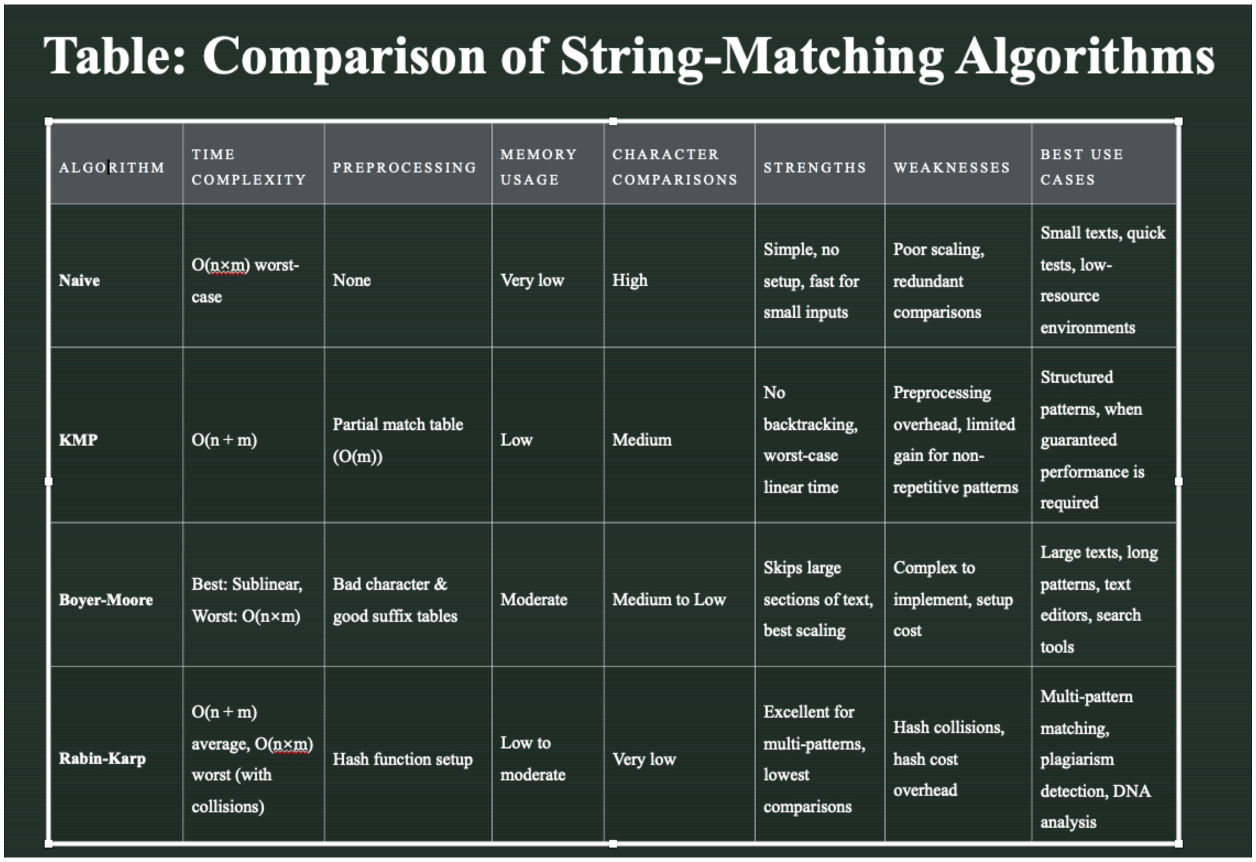


Table: Comparison of String-Matching Algorithms

ALGORITHM	TIME COMPLEXITY	PREPROCESSING	MEMORY USAGE	CHARACTER COMPARISONS	STRENGTHS	WEAKNESSES	BEST USE CASES
Naive	O(n×m) worst-case	None	Very low	High	Simple, no setup, fast for small inputs	Poor scaling, redundant comparisons	Small texts, quick tests, low-resource environments
KMP	O(n + m)	Partial match table (O(m))	Low	Medium	No backtracking, worst-case linear time	Preprocessing overhead, limited gain for non-repetitive patterns	Structured patterns, when guaranteed performance is required
Boyer-Moore	Best: Sublinear, Worst: O(n×m)	Bad character & good suffix tables	Moderate	Medium to Low	Skips large sections of text, best scaling	Complex to implement, setup cost	Large texts, long patterns, text editors, search tools
Rabin-Karp	O(n + m) average, O(n×m) worst (with collisions)	Hash function setup	Low to moderate	Very low	Excellent for multi-patterns, lowest comparisons	Hash collisions, hash cost overhead	Multi-pattern matching, plagiarism detection, DNA analysis

6.3 Practical Implications

The experimental results yield several practical implications for algorithm selection in real-world applications:

Text Editors and Search Functions: Boyer-Moore's performance characteristics make it ideal for text editors and search utilities, explaining its widespread adoption in these applications. Its ability to skip large portions of text is particularly valuable when searching through large documents.

DNA Sequence Analysis: For bioinformatics applications involving longer patterns, Boyer-Moore and Rabin-Karp would be preferred. Rabin-Karp's efficiency with multiple patterns makes it particularly suitable for genome matching with multiple query sequences.

Web Search Engines: The choice depends on the specific requirements, but Boyer-Moore offers a good compromise between preprocessing overhead and matching efficiency for typical web search scenarios.

Runtime Constraints: In applications with strict memory or processing limitations (like embedded systems), the Naive algorithm might be preferred despite its theoretical inefficiency, as it requires no additional memory and minimal preprocessing.

Development Considerations: For rapid prototyping or when matching performance isn't critical, Naive offers the advantage of simplicity and ease of implementation, with reasonable performance for many common scenarios.

These implications demonstrate that algorithm selection should be based not only on performance metrics but also on the specific constraints and requirements of the application context.

6.4 Limitations and Future Work

This study focused on single pattern matching in text with certain characteristics. Future work could:

1. Extend testing to much larger texts (>1MB) to further explore scaling behavior
2. Implement and test multi-pattern versions of algorithms, particularly Rabin-Karp
3. Explore hybrid approaches that combine the strengths of multiple algorithms
4. Test with specialized text types like DNA sequences or programming code
5. Examine memory usage and cache behavior in addition to comparisons and execution time
6. Implement optimizations like the good suffix rule for Boyer-Moore

These extensions would provide a more comprehensive understanding of string-matching algorithm performance across an even wider range of scenarios.

7. Conclusions & Recommendations

This study compared four string matching algorithms—Naive, Knuth-Morris-Pratt (KMP), Boyer-Moore, and Rabin-Karp—to evaluate their performance in different situations. By running experiments on patterns of different lengths and text sizes, we were able to gain valuable insights into how each algorithm behaves in practice, and how this compares to what theory tells us.

7.1 Key Insights

Theory vs. Real-World Performance

Although KMP is known for its linear-time complexity in theory, it did not always perform the fastest in practice. The extra time spent on preprocessing made it slower in some cases. On the other hand, the Naive algorithm, which is considered less efficient in theory, often gave faster results for short patterns. This shows that real-world performance depends on more than just the algorithm's big-O notation—it also depends on things like setup time and how the code is implemented.

Pattern Length and Structure Matter

All four algorithms reacted differently to changes in the pattern. Boyer-Moore and Rabin-Karp performed better when patterns were longer, while Naive and KMP were more consistent regardless of pattern length. The content of the pattern—especially if it had repeating characters—also affected the performance. This means the structure of the pattern, not just its length, plays an important role.

Execution Time vs. Number of Comparisons

One surprising finding was that doing fewer character comparisons didn't always lead to faster execution. For example, Rabin-Karp made far fewer comparisons than the Naive algorithm—sometimes 90–95% less—but it still didn't always run faster, because hashing adds extra processing time. This shows that measuring comparisons alone is not enough; we also need to consider other operations that the algorithm performs.

No Algorithm Works Best for Everything

There was no single algorithm that outperformed all others in every situation. Each had its own strengths and weaknesses depending on the pattern and text size. This means choosing the best algorithm depends on the specific task, and we shouldn't rely only on theoretical performance when making that choice.

7.2 Recommendations

Based on the results, here are some practical suggestions for when to use each algorithm:

Short Patterns and Small Texts

The **Naive** algorithm is a good option for simple tasks. It works well when the text and pattern are both short and doesn't need any setup time. It's suitable for small programs, quick searches, or situations where performance isn't critical.

Long Patterns and Large Texts

Boyer-Moore was the fastest in most tests involving long patterns. It is a strong choice for applications like file searching, word processors, or systems that need to handle large documents efficiently.

Searching for Many Patterns at Once

Although not directly tested, **Rabin-Karp** is well known for handling multiple pattern searches efficiently. This makes it useful for tasks such as plagiarism checking, genetic sequence analysis, or spam detection.

Stable Performance in All Cases

KMP is useful when I need consistent and predictable performance, regardless of the input. This is especially important in real-time systems or cases where the input might be crafted to slow down other algorithms.

When I Need a Quick and Simple Solution

The **Naive** algorithm is easy to understand and quick to implement. It is a good choice for learning, testing, and building basic programs where string matching is not the main focus.

7.3 Final Thoughts

This project showed that testing algorithms with real data is just as important as understanding their theoretical background. While big-O notation is helpful, it doesn't always predict how an algorithm will perform in practice. Many other factors—like preprocessing time, hashing costs, and pattern structure—can affect performance.

By combining theoretical understanding with real-world testing, developers and researchers can make better decisions when choosing algorithms for their projects. The project successfully achieved its primary goal of understanding when each string matching algorithm performs best, providing clear guidance for algorithm selection based on specific use cases and requirements. In future work, it would be helpful to explore how these algorithms perform when matching many patterns at once, handling huge datasets, or combining techniques for better overall results.

8. References

Baeldung. (2021). String Matching Algorithms in Java. Retrieved from https://www.baeldung.com/java-pattern-matching

Boyer, R. S., & Moore, J. S. (1977). A fast string searching algorithm. Communications of the ACM, 20(10), 762-772. https://doi.org/10.1145/359842.359859

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms (3rd ed.). MIT Press.

GeeksforGeeks. (2023). Naive Algorithm for Pattern Searching. Retrieved from https://www.geeksforgeeks.org/naive-algorithm-for-pattern-searching/

GeeksforGeeks. (2023). KMP Algorithm for Pattern Searching. Retrieved from https://www.geeksforgeeks.org/kmp-algorithm-for-pattern-searching/

GeeksforGeeks. (2023). Boyer-Moore Algorithm for Pattern Searching. Retrieved from https://www.geeksforgeeks.org/boyer-moore-algorithm-for-pattern-searching/

GeeksforGeeks. (2023). Rabin-Karp Algorithm for Pattern Searching. Retrieved from https://www.geeksforgeeks.org/rabin-karp-algorithm-for-pattern-searching/

Gusfield, D. (1997). Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology. Cambridge University Press.

Horspool, R. N. (1980). Practical fast searching in strings. Software: Practice and Experience, 10(6), 501-506.

Knuth, D. E., Morris, J. H., & Pratt, V. R. (1977). Fast pattern matching in strings. SIAM Journal on Computing, 6(2), 323-350. https://doi.org/10.1137/0206024

Karp, R. M., & Rabin, M. O. (1987). Efficient randomized pattern-matching algorithms. IBM Journal of Research and Development, 31(2), 249-260.

OpenDSA. (2023). String Pattern Matching. In OpenDSA Data Structures and Algorithms Online Book. Retrieved from https://opendsa-server.cs.vt.edu/ODSA/Books/Everything/html/StringMatching.html

RosettaCode. (2023). String Matching. Retrieved from https://rosettacode.org/wiki/String_matching

Sedgewick, R., & Wayne, K. (2011). Algorithms (4th ed.). Addison-Wesley Professional.

Tarhio, J., & Ukkonen, E. (1993). Approximate Boyer-Moore string matching. SIAM Journal on Computing, 22(2), 243-260.

9. Appendices

Appendix A: Implementation Details

The complete source code for this project is available in the accompanying file string-matching-algorithms.zip.

The link from giHub: https://github.com/Linda-Miao/tcss343-string-match

This archive contains:

- src/StringMatcher.java: The common interface implemented by all algorithms
- src/algorithms/NaiveMatcher.java: Implementation of the Naive algorithm
- src/algorithms/KMPMatcher.java: Implementation of the Knuth-Morris-Pratt algorithm
- src/algorithms/BoyerMooreMatcher.java: Implementation of the Boyer-Moore algorithm
- src/algorithms/RabinKarpMatcher.java: Implementation of the Rabin-Karp algorithm
- src/tests/TextLoaderTest.java: The testing framework used for all experiments
- data/: Directory containing test text files

All implementations follow the same interface design pattern, with each algorithm implementing the StringMatcher interface that defines two key methods:

- int[] findMatches(String text, String pattern): Returns all occurrences of the pattern in the text
- long getComparisons(): Returns the number of character comparisons performed

The code is extensively documented with comments explaining the algorithm mechanics, implementation choices, and instrumentation approach for measuring performance.

Appendix B: Test Data

B.1 Small Text Sample (First 150 characters)

This is a small test file for string matching algorithms.

We will search for patterns in this text.

The naive algorithm should find all occurrences of patterns.

B.2 Test Patterns

Pattern	**Length**	**Description**
"the"	3	Common word
"and"	3	Common word
"in"	2	Common preposition
"algorithm"	9	Technical term
"ababababab"	10	Repeating pattern
"implementation of string matching algorithms"	44	Long phrase
"theXYZ"	6	Rare pattern
"abcdef"	6	Sequential characters
"fedcba"	6	Reverse sequential

Appendix C: Raw Performance Data.

C.1 Character Comparison Counts

Pattern	Naive	KMP	Boyer-Moore	Rabin-Karp
"the"	68,964	62,772	26,330	3,535
"algorithm"	66,952	62,778	8,941	706
"ababababab"	66,666	62,781	6,736	663
"implementation..."	66,367	62,817	3,958	628
"theXYZ"	70,016	62,775	11,354	833
"and"	67,430	62,772	23,821	2,619
"for"	64,096	62,772	23,846	992
"abcdef"	66,668	62,775	12,210	596
"fedcba"	64,034	62,775	12,126	572

C.2 Execution Times (nanoseconds)

Pattern	Naive	KMP	Boyer-Moore	Rabin-Karp
"the"	1,405,490	247,167,880	34,102,364	38,002,868
"algorithm"	3,275,762	22,747,241	10,341,968	17,301,278
"ababababab"	810,103	8,634,898	1,586,764	6,111,072
"implementation..."	502,848	12,461,139	942,049	10,655,196
"theXYZ"	972,019	10,724,927	5,621,546	13,282,743
"and"	930,036	10,317,783	3,703,565	11,225,492
"for"	842,220	32,836,759	2,978,077	5,601,425
"abcdef"	1,398,066	11,129,443	13,136,863	4,569,331
"fedcba"	603,233	17,842,726	5,765,513	7,519,322